HiLA User Guide

UNICORE Team

Document Version:	1.0.0
Component Version:	2.3.0-rc
Date:	16 04 2012

This work is co-funded by the EC EMI project under the FP7 Collaborative Projects Grant Agreement Nr. INFSO-RI-261611.

Contents

1 Preface **1**

2 Current Usage **1**

3 Concepts **2**

3.1 Location and Resources 2

3.2 Programmatic use 2

3.3 Exceptions 3

4 The Grid API **3**

4.1 Grid 5

4.2 Site 6

4.3 Storage 6

4.4 Task 6

4.5 Job 7

4.6 Reservation 7

4.7 File 7

5 Development **7**

5.1 JobModel 8

6 Implementations **9**

6.1 UNICORE 6 10

6.2 EMI Execution Services 12

6.3 GridFTP 13

7 References **14**

1 Preface

Highly dynamic features and security concerns often make it difficult for programmers to easily develop clients accessing Grids. There is a lot of programming overhead involved for each call to a Grid resource, although many of these tasks are structured similarly. A universal, high-level API can alleviate the need to perform the tedious and repetitive tasks of accessing Grid resources and let the developer focus on the purely functional aspects of programming. The extra amount of work necessary to access a site is hidden behind a concise interface.

Another advantage of a cleanly defined API is the opportunity to implement for multiple back-end Grid environments. The HiLA API currently supports two implementations, one for UNICORE 5 and more recently another one for the Web services based UNICORE 6. An implementation for OGSA-BES is in a development state. Additional backend implementations are conceivable to support other Grid middlewares in the same uniform way.

Rather than designing a task focused API, HiLA takes a resource oriented approach to the definition of the Java interfaces. The linkages between the resources on the Grid are approached in a consistent manner. A resource is Locatable and possesses a Location. As such, there is a generic mechanism in place for navigating the linkages. For example, a Site has a number Storages, a Storage contains a number of Files and a Site references its running Tasks. So different underlying Grid infrastructures become browsable in a uniform way. Analysing and using Grid resources can then be as easy as browsing a UNIX file system.

The API uses a Factory mechanism to dynamically load particular implementations. Multiple implementations of the API can co-exist within the lifetime of a program execution, and it encourages late binding to a specific implementation, selectable at runtime. Location of the HiLA resources are based on URI's, where the scheme is used to select a particular implementation.

This document describes the HiLA API from a user's point of view. Of course, users of APIs are developers. There is a separate developer guide available for those, who want to develop additional implementations of the abstract interfaces.

Originally, HiLA was named Roctopus, which is why you will find references to this name as well [Hagemeier2007] [roctopus_cgw05].

2 Current Usage

HiLA is a convenient toolkit for building web interfaces or command line environments for Grids, and is used by the DESHL suite of tools developed in the JRA7 activity of the DEISA project. In the A-WARE project, we are using HiLA as the basis for a Grid agent component to be plugged into a service bus, which is then used as for building higher-level orchestration services for Grids.

HiLA is an API for accessing Grid resources of different middleware in a consistent manner. It has been implemented for UNICORE versions 5 and 6, and an OGSA-BES implementation is also available. HiLA is in use in various places in different Grid access libraries. The DEISA Services for Heterogeneous management Layer (DESHL) has been implemented using HiLA

right from the start. This severely eased the shift to UNICORE 6, as HiLA provides the same interface for both middlewares. In the German AeroGrid project [aerogrid], a JavaGAT adaptor has been implemented on top of HiLA, thus enabling the use of UNICORE 5 and UNICORE 6 through GAT and this adaptor.

3 Concepts

3.1 Location and Resources

The key concept in HiLA is that every Resource can be referenced by a URI.

The generic URI structure in HiLA (for Grids) looks like this:

```
<scheme>:/sites/<site-name>/tasks/<task-id>/wd/files/<file-name>
<scheme>:/sites/<site-name>/storages/<storage-name>/files/<file- ↩
    name>
```

The `<scheme>` placeholder is the implementation specific part of the URI. In real scenarios, it will be something like `unicore6`, `unicore5` or `ogsa`.

The actual URI is pattern matched against the patterns of registered ResourceTypes.

Staying with the Grid example, `<site-name>` will be the name of the site, as it has been discovered by the configuration of HiLA Grid API. `<task-id>` is the id of the task, which is generated by the middleware. This enables us to located particular tasks only from this reference. The same holds for `<storage-name>`, where all storages exposed by the middleware at the site under consideration will be valid values for `<storage-name>`.

This follows a regular pattern, where collections and their contents alternate in the URI structure. Collections are `sites`, `tasks`, `storages`, and `files`. While all of these do have equivelents in the Java interfaces, they are less important, because they do not offer mor functionality than any other Locatable artifact.

The concept of artifacts being locatable is generic and extensible. Which artifacts in the Grid can be referenced by HiLA, will be subject of the following sections.

3.2 Programmatic use

```
Location loc = new Location("unicore6:/sites/");
Resource res = loc.locate();
```

As you can see, a `Location` has a `locate()` method, with which you can locate the `Resource` behind this `Location`. Of course, it is only this simple when dealing with generic `Resources`. Most of the time, however, you will want to deal with specific Resources, on which you can call certain actions. Have a look at the following example:

```
Location loc = new Locatin("unicore6:/sites/DEMO-SITE_201005231038 ↩
    ");
if(loc.isLocationOfType(Site.class) {
  Site site = (Site) loc.locate();
}
```

Another option would be (assuming `loc` is still the same):

```
Resource res = loc.locate();
if(res instanceof Site) {
  Site site = (Site) res;
}
```

3.3 Exceptions

A base set of Exceptions have been defined to represent certain error or exceptional conditions. All of them are derived from `HiLAException`.

- HiLAException
  - HiLAAccessDeniedException
  - HiLACannotContactSiteException
  - HiLAFactoryException
  - HiLAIdentityException
  - HiLALocationPatternException
  - HiLALocationSyntaxException
  - HiLANotImplementedException
  - HiLANotSupportedException
  - HiLAResourceAlreadyExistsException
  - HiLAResourceNotFoundException

4 The Grid API

The HiLA API with its notions of `Location` and `Resource` is very generic and merely allows to create a navigable hierarchy of `Resources` that have `Locations`.

The basic artifacts, which can be accessed by HiLA and referenced via HiLA locations, are:

- Grid
- Site

- Storage
- Task
- Reservation
- File

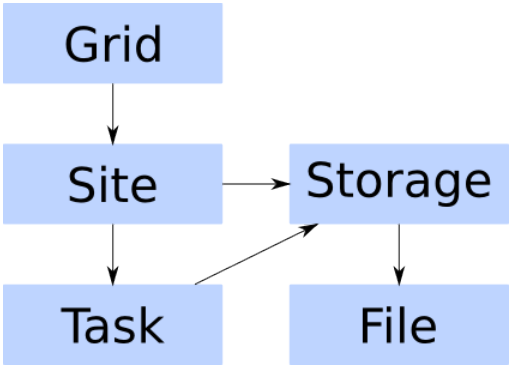

Figure 1: Architectural overview of HiLA

The HiLA Architecture figure shows the interrelationships between the various artifacts. A Grid is made up of a number of sites, which in turn contain tasks and storages. Storages contain files, which can be imported to, exported from or transferred among storages. Tasks also have a storage, which is their working directory. One can thus access the working directory of a task at any time, if the underlying implementation supports it.

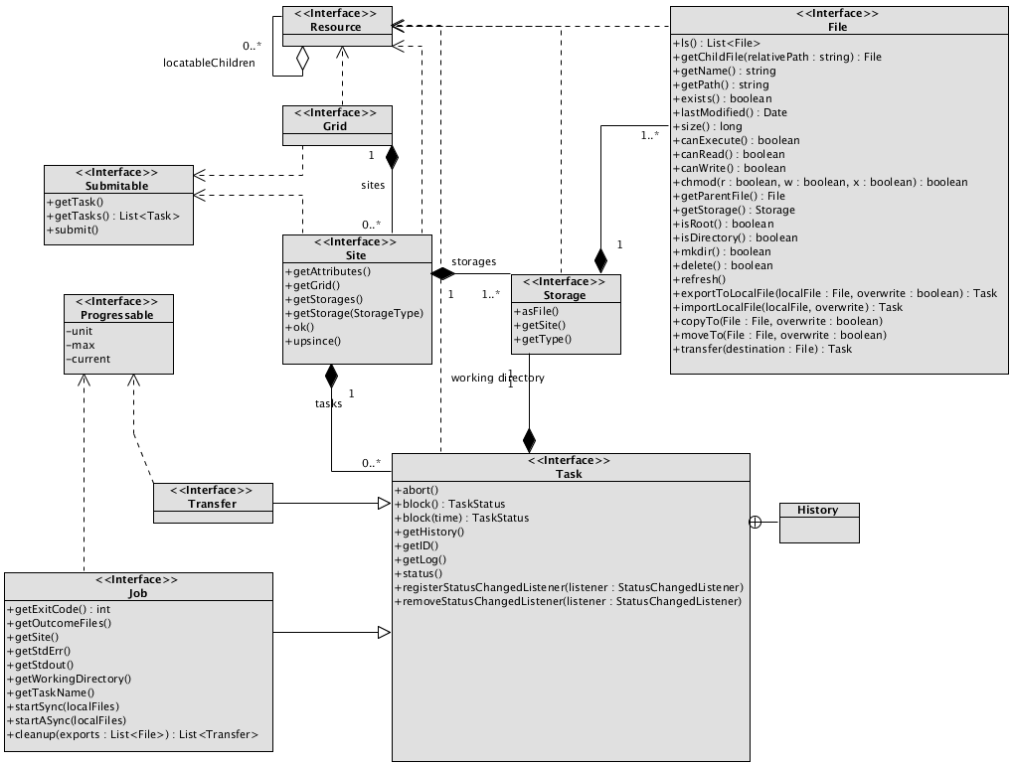

Figure 2: UML class diagram of HiLA API

The UML class diagram shows the main classes within the abstract HiLA API. An implementation of HiLA has to implement the interfaces as appropriate and support the given operations. Operations on the interfaces will be described in the following.

4.1 Grid

Grid locations are structured like `<scheme>:/`. For example, a grid location for the UNICORE 6 implementation of HiLA would look like `unicore6:/`. As shown in the above figure and more specifically in the UML diagram, a `Grid` consists of an arbitrary number of `Sites`. You can query the Sites belonging to a `Grid` by calling the `getAllSites()` method, which returns a List of `Sites`. If you know the `Location` of an entity, to which you want to gain access, you can `locate()` it. The `locate()` method returns a `Locatable`, which you may have to cast to the appropriate type.

4.2 Site

Site locations are structured like `<scheme>:/sites/<site-name>` A Site location for the UNICORE 6 implemenation looks like `unicore6:/sites/Alpha-Site`. Sites implement both the Locatable and Submitable interfaces. While Locatable has been explained before, the Submitable interface is used for anything that one can submit jobs to. Hence, it implements the methods `submit()`, `getTasks()`, and `getTask()`.

4.3 Storage

Storage locations are structured like

`<scheme>:/sites/<site-name>/storages/<storage-name>`.

There is a special twist to Storage Locations, as a Task also has a Storage, where its execution takes place. Thus, a Storage Location can also look like

`<scheme>:/sites/<site-name>/tasks/<task-name>/wd`

where `wd` means working directory. Storages belonging to Sites can be retrieved by calling `getStorages()` on a Site object. From a Task, one can directly jump into the working directory by calling `getWorkingDirectory()` on the Task, which actually returns a File, but that is what you want anyway.

Examples of Storage Locations for the UNICORE 6 implementation would be

`unicore6:/sites/Alpha-Site/storages/home` and

`unicore6:/sites/Alpha-Site/tasks/ \`
`79258434-f295-11dd-b58a-001f160cb28c/wd`.

4.4 Task

A `Task` is something that can run concurrently with the rest of the code, e.g. a file transfer or remote `Job`. `Task`s can be monitored.

Task locations are structured like `<scheme>:/sites/<site-name>/tasks/<task-name>`. An example Task Location for the UNICORE 6 implementation looks like this `unicore6:/sites/ \`
`Alpha-Site/tasks/79258434-f295-11dd-b58a-001f160cb28c`.

One can register listeners on `Tasks` that listen for changes of its status.

```
Task t = ...;
t.registerStatusChangedListener(new StatusChangeListener() {
  public void statusChanged(StatusChangedEvent e) {
    System.out.println("Status of Task: " ++ e.getTask(). ↩
        getLocation() ++ " changed to " ++ e.getStatus());
  }
});
```

4.5 Job

A specialization of the generic `Task` is a `Job`, which has an exit code, outcome files, a working directory etc. Unlike ordinary `Task`s, `Job`s have a `Storage` representing the `Job`s' working directory, which can be browsed like any other `Storage`.

4.6 Reservation

Starting with version 2.2, `Site`s implement the `Reservable` interface, which allows for creating reservations based on `ResourceDescription`s. Since not all `Site`s or even Grid middleware support this, it is very likely that calling the `reserve` method throws a `HiLANotImplementedException` or `HiLANotSupportedException`.

`Job`s can be run inside `Reservation`s.

4.7 File

File locations are structured like

`<scheme>:/sites/<site-name>/storages/<storage-name>/files/<file-name>`.

Again, as Tasks have associated Storages as well, and these Storages also contain Files, an alternative structure of a File's location is

`<scheme>:/sites/<site-name>/tasks/<task-name>/wd/files/<file-name>`.

Example File Locations for the UNICORE 6 implementation look like the following:

`unicore6:/sites/Alpha-Site/storages/home/files/.bashrc` and

`unicore6:/sites/Alpha-Site/tasks/\`
`79258434-f295-11dd-b58a-001f160cb28c/wd/files/stdout`.

5 Development

If you would like to develop using the HiLA API, you will most likely start with the Location of a `Resource` serving as an entry point into the `Resource` hierarchy of your problem domain. It could be something as simple as a pure scheme name appended with »:/«, such as `unicore6:/`.

In code, this would look something like this:

```
Location loc = new Location("unicore6:/sites/DEMO-SITE_201004261103 ↩
    ");
if(loc.isLocationOfType(Site.class)) {
  Site site = (Site) loc.locate();
  // do something with that site
}
```

As you can see, you can start from a location and have it locate() the resource. Generally, the locate() method returns objects of type `eu.unicore.hila.Resource` or one of its decendants. You can already query the `Location`, if it points to a certain resource type. That way, it is not necessary to locate (and instantiate) the actual resource, if you already know that the `Location` points to the wrong resource type.

If you want to be even more sure about the type of `Resource` returned by `locate()`, you can check it using the `instanceof` operator.

```
Resource res = loc.locate();
if(res instanceof Site) {
  Site site = (Site) res;
}
```

However, this should be obsolete, if the implementation of the API does not misbehave.

5.1 JobModel

Starting with version 2.3.0, HiLA supports a more sophisticated JobModel that is supposed to better support the multitude of job description languages that are used by different Grid middlewares. Before that version, the `Submitable.submit()` merely consumed an object that the implementation had to know what to do with.

The benefit of consuming a real abstraction of the job description, the job model, it is fully up to the implementation to map it to the correct description language. Before, it was partly up to the user (developer) to know what job description could be consumed by a particular implementation and thus to also be aware against which implementation the code was written.

While the old method still remains, but is deprecated, the new method should be used from now on. It will be described in the following.

The job model is centered around the `JobModel` class, which is an abstraction of common job description languages. The model itself is agnostic of any particular job description languages, i.e. JSDL [?] or EMI ADL [emies].

5.1.1 Applications or Executables

All Grid middlewares support the concept of running an explicit executable on the Grid site. This can be set in the job model as an executable. Some middlewares support the concept of applications that are mapped down to an executable with possibly additional arguments. These two concepts are treated as mutually exclusive by the job model implementation. Thus, setting an executable unsets an application that has previously been set and vice-versa.

5.1.2 Arguments and Environment

Arguments are treated as a list of commandline arguments. The existing list can be extended by `add`ing arguments. Another way to set arguments it to `set` the list in a single operation.

The environment is a list of environment variables and their corresponding values. Being a list, this is treated in the same way as arguments, one can either `add` additional variables or `set` the entire list in one operation.

5.1.3 Data transfer

Most job description languages only know about third-party transfers (as opposed to client-server transfers) for job submission. There's a good reason for this, as the job description can be passed on to the server side as is and processed there.

On the other hand, it can be very handy to be able to refer to local files for copying them into the job's working directory before the job starts and retrieving output files from the same directory after the job has finished.

The job model captures this situation by distinguishing these two scenarios. It provides `StageIn` and `StageOut` classes for the third-party scenario and `Import` and `Export` classes for client to server transfers.

5.1.4 Resources

...

6 Implementations

The following table shows the relationships among the different implementations of HiLA APIs. At the base, there is hila-api, which is generic and does not know anything about Grid or other specific resources. It merely provides the basis for implementing resource types and giving them structured locations.

On the next level, there are hila-grid-api and hila-other-api, where the latter just represents any other API that is based on hila-api.

<table>
<tr><td>hila-unicore6</td><td rowspan="2"></td></tr>
<tr><td>hila-grid-common</td></tr>
<tr><td>hila-grid-api</td><td>hila-other-api</td></tr>
<tr><td colspan="2">hila-api</td></tr>
</table>

6.1 UNICORE 6

6.1.1 Additional Capabilities

Orchestrator

The UNICORE 6 implementation supports the service orchestrator, which is used as a broker to submit jobs to `Sites` matching the resource requirements given in the job description.

6.1.2 Configuration

Configuration for this implementation is expected in the user's home directory under ~/.hila2/unicore6.properties. The format of this file is the following

Basic Configuration

This configuration should be used whenever you can use your credentials directly, i.e. you have the keystore with your private key at hand. The next section will describe what you have to do if you configure HiLA to be used as an agent, e.g. inside a portal, and users have issued trust delegations, which allow the agent to act on their behalf. Another use case for the advanced configuration below is when you have multiple identities.

```
hila.unicore6.registries = https://localhost:8080/DEMO-SITE/ ↩
    services/Registry?res=default_registry

hila.unicore6.keystore = /home/demo/.hila2/demo.jks
hila.unicore6.alias = demo user
hila.unicore6.password = the!user

# The following properties are optional
hila.unicore6.keystoretype = jks
hila.unicore6.truststore = /home/demo/.hila2/demo.jks
hila.unicore6.truststorepassword = the!user
hila.unicore6.truststoretype = jks
```

with the following meanings of the elements.

hila.unicore6.registries
: A space separated list of UNICORE 6 registries. These registry URLs should be provided to you by administrators of your Grid infrastructures.

hila.unicore6.keystore
: A Java KeyStore in PKCS12 or JKS format. The path may be relative to the configuration file.

hila.unicore6.alias
: The alias of the key inside the keystore.

hila.unicore6.password
Password to open the KeyStore.

hila.unicore6.keystoretype
Type of keystore: JKS or PKCS12. Defaults to JKS

hila.unicore6.truststore
A truststore with certificates to be used for verification. If not given, keystore will also be used as truststore. The path may be relative to the configuration file.

hila.unicore6.truststorepassword
Password for truststore. If not given, the keystorepassword will be used.

hila.unicore6.truststoretype
Should be JKS, which is the default.

Optional Configuration

```
hila.unicore6.timeout = 5000
hila.unicore6.refresh = 100000
hila.unicore6.separatetss = false
```

hila.unicore6.timeout
Timeout in ms for site discovery.

hila.unicore6.refresh
Time in ms until the sites cache will be refreshed from the registries. Until then, previously discovered sites will be returned.

hila.unicore6.separatetss
Whether to represent multiple target system resources belonging to a single site as separate sites, appended with the creation time to distinguish them. The value of this property should be `true` or `false`. The default value is `false`.

Multi-User Configuration

This configuration can be used for multiple users, either in case you have multiple identities and need to distinguish among them, or when you are an agent that acts on other user's behalf. In the latter case, users will have issued trust delegations.

Security Profiles

```
hila.unicore6.keystore = /home/demo/.hila2/demo.jks
hila.unicore6.alias = demo
hila.unicore6.password = the!user
```

```
# Demo 1
hila.unicore6.profile.demo1.keystore = /home/demo/.hila2/demo1.jks
hila.unicore6.profile.demo1.alias = demo1
hila.unicore6.profile.demo1.password = the!user

# Demo 2
hila.unicore6.profile.demo2.keystore = /home/demo/.hila2/demo2.jks
hila.unicore6.profile.demo2.alias = demo2
hila.unicore6.profile.demo2.password = the!user
```

The multi-user configuration depends on the existance of an implementation of the `Config` interface. This can be configured via the `hila.unicore6.config` property. The default value for this property is

`eu.unicore.hila.grid.unicore6.security.SimpleProfileConfig`, which can use profiles as configured above. If you want to use any of the configured profiles, then the profile name needs to be added to the HiLA `Location` of the resources, e.g. `unicore6:/demo1@sites` or `unicore6:/demo2@sites` will use the keystores as configured above. Note that this does not have anything to do with trust delegation, each of the profile configurations will be used as primary identities to access Grid resources.

Trust Delegation

In order to use trust delegation, you need to use a different implementation of the `Config` interface. The unicore.properties file should contain the following line:

```
hila.unicore6.config = eu.unicore.hila.grid.unicore6.security. ↩
    TDConfig
```

`TDConfig` requires SAML assertions in the directory `~/.hila2/saml-assertions`. SAML assertions can be generated e.g. using the UNICORE Commandline Client (UCC) and its `issue-delegation` command. HiLA as an API does not offer this functionality at the moment. Trust delegation in HiLA is usually used in locations where the user credentials needed for creating the delegation are not available. Thus, HiLA as the receiver of such SAML assertions does not require this functionality.

6.2 EMI Execution Services

The EMI Execution Services (EMI-ES) specification [emies] is an agreement among European middleware providers Arc, gLite, and UNICORE, to offer a common interface to their compute elements (CEs), which is the equivalent of a HiLA Site.

Starting with version 2.3.0, HiLA supports this specification and can send jobs to EMI-ES endpoints.

6.2.1 Configuration

Analogous to the UNICORE 6 configuration file, which is expected to reside in ${user.home}/.hila2/unicore6.properties, the EMI ES configuration file is expected in the same directory as ${user.home}/.hila2/emi-es.properties.

```
hila.emi.es.endpoint.base.1 = https://localhost:8080/DEMO-SITE/ ↩
    services

hila.emi.es.keystore = demo.jks
hila.emi.es.alias = demo user
hila.emi.es.password = the!user
```

Configuration properties have the following meanings.

hila.emi.es.endpoint.base.n
: A base URL for the EMI ES services. The implementation expects these services to be named ActivityInfoService, ActivityManagementService, CreateActivityService, DelegationService, and ResourceInfoService. Thus, a full service URL would result in https://localhost:8080/DEMO-SITE/services/ActivityInfoService

hila.emi.es.keystore
: A Java KeyStore in PKCS12 or JKS format. The path may be relative to the configuration file.

hila.emi.es.alias
: The alias of the key inside the keystore.

hila.emi.es.password
: Password to open the KeyStore.

6.2.2 Restrictions

The EMI ES specification is only about compute services. It allows for submitting jobs and monitoring their status. Additionally, it allows for retrieving information about `Sites`.

6.3 GridFTP

The GridFTP implementation allows for accessing GridFTP services using ordinary gsiftp:/ URLs.

6.3.1 Restrictions

GridFTP does not currently use any additional configuration. It consumes direct gsiftp:/ URLs like any ordinary GridFTP client. Therefore, you will need a local Grid proxy certificate initialized with grid-proxy-init. Future versions of this implementation will be capable of using the user's certificate directly, as is the case for the other implementations.

7 References

[1] [Hagemeier2007] Björn Hagemeier and Roger Menday and Bernd Schuller and Achim Streit. "A Universal API for Grids". Bubak, M., Turala, M. & Wiatr, K. (ed.) Cracow Grid Workshop '06 Academic Computer Centre CYFRONET AGH, 2007, pp. 312-319 ISBN 83-915141-7-X.

[2] [roctopus_cgw05] Menday, R.; Kirtchakova, L.; Schuller, B. \& Streit, A. "An API for Building New Clients for UNICORE" Proceedings of the Cracow Grid Workshop '05, 2005

[3] [aerogrid] http://www.aero-grid.de/

[4] [emies] EMI Execution Service Specification - https://twiki.cern.ch/twiki/-pub/EMI/EmiExecutionService/EMI-ES-Specification_v1.07.odt